Are Ownership Types Reaching the World Yet?

Patrick Lam
University of Waterloo

@uWaterlooSE
https://patricklam.ca

July 2016

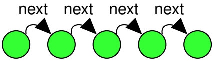
Wrigstad & Clarke, IWACO ’11:
“Is the World Ready for Ownership Types?
Is Ownership Types Ready for the World?”

Outline

1. Goals of Ownership Types
2. Ownership Types in the World:
    - Rust
    - Other Languages

Why Ownership Types?

Let's motivate with a linked list.



Clearly, next field should be private:

```
class Node {
  private:
    int data;
    Node * next;
};
```

Are we good?

i.e. protected against external changes?

```
class Node {
  private:
    int data;
    Node * next;
};
```

We like to think so.

- Java: absolutely.
- C/C++: yes, with caveats;
  good enough for most.

OK, so why Ownership Types?

```
class C {
  private:
    std::vector<int> items;
}
```

If we expose the `items` vector,
then the recipient can change it via an alias!

Problem: Uncontrolled aliasing is
hard to deal with.

Solution: Restrict aliasing!

Goals of Ownership Types

"Bugs due to unintentional aliases are
notoriously difficult to track down."

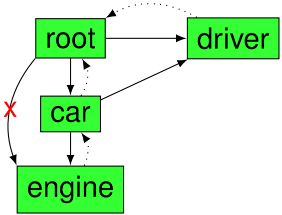
"Dealing with aliasing...
a key research issue for OOP."

Why? "shared mutable state" & "stable object identity".

— from "Ownership Types: A Survey", Clarke et al.

What Ownership Types Do

Enable developers to enforce aliasing constraints between components.



(Credit: “Flexible Alias Protection” by Clarke, Potter and Noble)

Guarantees Provided by Ownership Types

Topological Organization:

structure heap into separate sub-heaps,
each of which has unity of purpose.

Encapsulation:

prevent non-local changes to shared state,
by controlling sharing & making access permissions visible.

Applications of Ownership Types

visualization & understanding
memory management
concurrency control
verification
security

Ownership Types in the World: Rust

Rust manual, 2016:

“This is the first of three sections presenting Rust’s ownership system. This is one of Rust’s most distinct and compelling features…

By Salisapants (Own work) [CC BY-SA 4.0], via Wikimedia Commons

“The United States and Great Britain are two countries separated by a common language.”

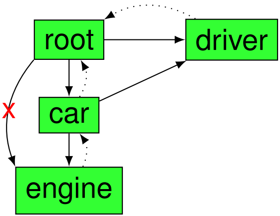
— apocryphally, George Bernard Shaw

By Luis2492 - Own work, GFDL, https://commons.wikimedia.org/w/index.php?curid=3947321

Applications

Clarke, Potter & Noble,
"Ownership Types for Flexible Alias Protection."

Representation containment;
owners-as-dominators.



Rust:

Memory safety
and management
Concurrency control

Rust's Safety Guarantees

No dangling pointers, via ownership types.

no resource leaks
no use-after-free
no reads of uninitialized values

No violation of declared lock policies.

(consequence of single-ownership)

No null pointer dereferences.

No buffer overruns.

Key Rust techniques

- single ownership of resources
- borrowing
- immutable objects

Guarantees verified at compile time.

Example: Rust Enforces Single Ownership

```
fn main() {
  let s = vec![1,2,3];
  let r = s;
  // s no longer owns the vec
  println!("s[0] is {}", s[0]);
}
```

The compiler refuses to compile this:

```
move.rs:4:26: 4:27 error: use of moved value: `s` [E0382]
move.rs:4    println!("s[0] is {}", s[0]);
```

Borrowing read-only references

```
fn borrowing(b: &Vec<i32>) {
    println!("b[0] is {}", b[0]);
}

fn main() {
    let s = vec![1,2,3];
    borrowing(&s);
}
```

Multiple active read-only references can exist.

While borrowed references alive, can't do writes.

Borrowing mutable references

```
fn borrowing_mutably(b: &mut Vec<i32>)
  { b[0] = 2; }

fn main() {
  let mut s = vec![1,2,3];
  {
    let t = &mut s;
    borrowing_mutably(t);
  }
  println!("s[0] is {}", s[0]);
}
```

A unique borrowed read-write reference can exist.

That reference has exclusive access to resource.

Implications of Single-Ownership System

Heap is a tree—no cross-references.

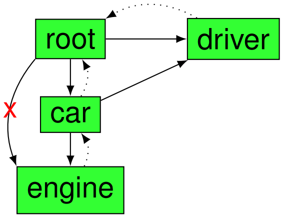
We get:
topological organization (heap is structured!)
encapsulation (no non-local changes!)

Resource management:
free when single owner goes out of scope.

Can implement a singly-linked list.

Limitations of Single-Ownership System

Can't express this:



Beyond Limitations

Two main options:

- RefCell (uniqueness checks at runtime).
- Unsafe raw pointers.

Rust's Applications of Ownership Types

~~visualization & understanding~~
memory management
concurrency control
~~verification~~
~~security~~

What Rust doesn't do

allow declaration of explicit owners / contexts;

support/enforce software architecture constraints;

allow multiple ownership.

Safe Rust and single/multiple ownership

Rust enforces single ownership of resources;
ownership can be borrowed (but must be returned).

Whenever multiple references exist,
no writes can occur.

How does Rust do:

By Salisapants (Own work) [CC BY-SA 4.0], via Wikimedia Commons

topological organization?

not necessarily organized, but no uncontrolled writes.

no way to specify the organization.

encapsulation?

yes, single-ownership,

plus immutable-by-default & marked mutable refs

By Luis2492 - Own work, GFDL, https://commons.wikimedia.org/w/index.php?curid=3947321

Ownership Types in the World: C++

C++11 includes features similar to Rust's.

- RAII
- unique_ptr
- shared_ptr

Unlike in Rust, smart pointers must be explicitly used.

C++: Immutability

Ownership types help deal with shared mutable state.

Prevention?
C++ provides **const** keyword.

Jon Eyolfson and I have studied **const** in C++ programs.

"C++ const and Immutability: An Empirical Study of Writes-Through-const"

ECOOP, Wednesday, 13:45.

C++: Resource Allocation Is Initialization (RAII)

Rust's resource management
is a generalization of RAII from C++.

```
void file_operation() {
  std::ofstream file(``example'');
  file << ``hi'' << std::endl;
  // no explicit close needed
}
```

When "file" goes out of scope, destructor closes.

Problems with RAII

RAII can't figure out when to free here:

```
using namespace std;
void string_operation_copy() {
  string* str1 = new string("!");
  string* str2 = str1;
  cout << str1 << endl; // (not allowed in Rust)
  cout << str2 << endl;
}
```

Reason: not clear who is the string's owner.

```
==27738== 32 bytes in 1 blocks are definitely lost in loss record 1 of 2
==27738==    at 0x4C2A23F: operator new(unsigned long) (vg_replace_malloc.c:334)
==27738==    by 0x400F02: string_operation_copy() (examples.cpp:7)
==27738==    by 0x4011C8: main (examples.cpp:29)
```

C++ RAII workaround 1: unique_ptr

```
using namespace std;
void string_operation_uniq() {
  auto str1 = make_unique<string>(``!'');
  auto str2 = move(str1);
  cout << *str1; // (segfault: null ptr deref)
  cout << *str2;
}
```

C++ implements move semantics (destructive reads)
by nulling on copy.

Similar to Rust's unique pointers, but fewer safety guarantees.

You can borrow in C++ (raw pointers);
but C++ has no borrow checker.

C++ RAII workaround 2: shared_ptr

```
using namespace std;
void string_operation_shared() {
  auto str1 = make_shared<string>(``!'');
  auto str2 = str1;
  std::cout << *str1 << endl;
  std::cout << *str2 << endl;
  // deallocated when refcount = 0
}
```

C++ allows (easily)

multiple mutable copies of shared object.

(subject to usual refcount limitations on cycles)

(weak_ptrs for cycles)

C++ Summary

Ownership types for:

~~visualization & understanding~~

memory management

(dynamic enforcement w/null derefs, fails fast)

~~concurrency control~~

~~verification~~

~~security~~

Other Languages: Scala

immutability preferred (“val”)

if you need shared mutable state, use actors:
    encapsulated shared mutable state
    send/receive immutable messages

Reduces need for ownership types
by discouraging mutability.

Other Languages

Go:

Has shared mutable state.
Encourages conventions for ownership.

Goroutines & event loops: similar to actors.

Dart:

No shared mutable state.
Encapsulates threads in “isolates”.

Swift, Clojure:

Values immutable, refs may change.

[swift] read/write queues
[clojure] all changes in a transaction or async

Conclusion

My take on practical ownership types in 2016:

Rust = usable simple compile-time ownership.

C++ = support for run-time ownership.

Other languages = alternatives to shared mutable state.

In summary:

✓ resource management applications

× software architecture applications